International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Web Vulnerabilities: Issues and Analysis

Dishant Modi[1], Karan Bhatt[2], Shivangkumar Patel[3], Viral Patel[4], Ashvinkumar Prajapati[5], Nitinkumar Raval[6], , Yogendra Tank[7]

Submitted: 29/01/2024 Revised : 12/03/2024 Accepted : 21/03/2024

Abstract: Pervasive and exploitable, software vulnerabilities pose a continuous threat to system security, empowering cybercriminals to disrupt operations, steal data, or compromise critical infrastructure. This paper leverages the OWASP Top 10, a recognized standard for web application security risks, to provide a comprehensive analysis of the top five most critical vulnerabilities. It delves into the technical details, potential consequences, and mitigation strategies for each of these vulnerabilities. The paper also offers a brief overview of the remaining OWASP Top 10 categories, equipping readers with a well-rounded understanding of prevalent web application security threats. By understanding these vulnerabilities and their analysis methods, organizations can proactively safeguard their web applications and enhance their overall cyber defense posture.

Keywords: *Open Web Application Security Project (OWASP), Vulnerability Disclosure, Vulnerability Research, Common Weakness Enumeration (CWE), Vulnerability Scoring Systems, Exploit Analysis, Exploit Development.*

1. Introduction

The digital world thrives on interconnected systems, but with this connectivity comes inherent vulnerabilities. A vulnerability is a weakness or flaw in the design, implementation, or operation of software that can be exploited by malicious actors to gain unauthorized access, steal data, disrupt operations, or cause other damage. These vulnerabilities can exist in various parts of a system, from the application layer down to the underlying hardware. The constant evolution of technology and the increasing sophistication of cyber-attacks necessitate a proactive approach to web application security. [1] The Open Web Application Security Project (OWASP) is a non-profit organization that plays a crucial role in web application security by providing free and open re-sources for developers and security professionals. A key contribution of OWASP is the OWASP Top 10, a widely recognized standard that identifies and categorizes the most critical web application security risks. The OWASP Top 10 is updated periodically to reflect the evolving threat landscape. Here's a list of the current Top 10 vulnerabilities (2021):

Table 1. OWASP TOP 10 Vulnerabilities

A01: 2021	Broken Access Control
A02: 2021	Cryptographic Failure
A03: 2021	Injection
A04: 2021	Insecure Design
A05: 2021	Security Misconfiguration
A06: 2021	Vulnerable and Outdated Components
A07: 2021	Identification and Authentication Failure
A08: 2021	Software and Data Integrity Failure
A09: 2021	Security Logging and Monitoring Failure
A10: 2021	Server-Side Request Forgery

2. Comprehensive analysis of the top ten most critical vulnerabilities listed in the OWASP Top 10.

2.1. A01:2021-Broken Access Control (BAC):

• **Broken Access Control (BAC):** Broken access control vulnerabilities arise when security mechanisms fail to restrict access to authorized users only. This allows attacker to bypass intended access controls and potentially view sensitive data, modify information, or perform unauthorized actions. For in-stance, a 2019 vulnerability in YouTube allowed attackers to access specific frames of videos marked as private. While a single frame might not reveal the

[1]*Department of Computer Science & Engineering (Data Science), Vishwakarma Government Engineering College, Chandkheda*
Email: dsmodi484@gmail.com
[2]*Assistant Professor, Department of Computer Engineering, Vishwakarma Government Engineering College, Chandkheda*
Email: kpbhatt@vgecg.ac.in
[3]*Assistant Professor, Department of Computer Engineering, Government Engineering College, Modasa*
Email: shivang.patel@gecmodasa.ac.in
[4]*Assistant Professor, Department of Computer Engineering, Government Engineering College, Sector-28, Gandhinagar*
Email: viralpatel@gecg28.ac.in
[5]*Assistant Professor, Department of Computer Engineering, Government Engineering College, Sector-28, Gandhinagar*
Email: ashvinkumar@gecg28.ac.in
[6] *Assistant Professor, Department of Computer Engineering, Government Engineering College, Sector-28, Gandhinagar*
Email: nitinraval@gecg28.ac.in
[7]*Assistant Professor, Department of Computer Engineering, Government Engineering College, Sector-28, Gandhinagar*
Email: yogendratank@gecg28.ac.in

entire content, an attacker could potentially reconstruct the video by requesting multiple frames sequentially. This highlights how seemingly minor access control flaws can have significant consequences, as users expect their private data to be truly inaccessible [2].

• **Insecure Direct Object Reference (IDOR):** Insecure Direct Object References (IDOR) represent a type of access control vulnerability where an attacker can access resources beyond their intended access privileges. This often occurs when an application exposes "direct object references," which are essentially identifiers pointing to specific objects on the server. These objects could be files, user accounts, bank accounts, or any other data entity. For example, consider a banking application where a user successfully logs in and is directed to a URL like https://xyz.com/account?id=11. This page displays the user's account details. However, if the application is vulnerable to IDOR, an attacker could potentially modify the id parameter in the URL (e.g., changing it to 22). A vulnerable application might then grant the attacker access to another user's bank information due to improper access control checks. This scenario underscores the importance of robust access control mechanisms to prevent unauthorized access to sensitive data.

To bridge the gap between theory and practice, we will examine a real-world example: As part of the practical exploration using TryHackMe [3], the first step is to access the provided website's login page.

In Fig.1. Login the user interface transitions to a new page displaying. Use parentheses to avoid ambiguities in denominators. Punctuate equations when they are part of a sentence, as in

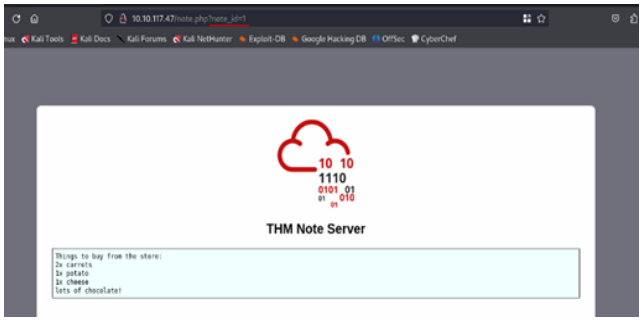

Fig. 1. Website homepage

Put id=0 (http://machine-ip/note.php?note_id=0) and you will get the flag as in Fig. 2:

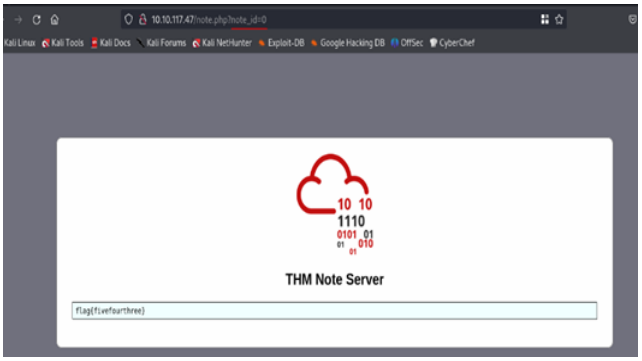

Fig. 2. Exploitation of Broken Access Control

By exploiting a BAC vulnerability like IDOR, an attacker could potentially gain unauthorized access to sensitive pages or data.

2.2. A02:2021-Cryptographic Failures

• **Cryptographic Failures and Sensitive Data Exposure:** Web applications often rely on cryptography to safe-guard sensitive user information, such as names, dates of birth, and financial data. However, crypto-graphic failures can arise due to the misuse or lack of proper implementation of encryption algorithms. These failures can lead to accidental data exposure, compromising user security and potentially violating privacy regulations.

• **Flat-File Database Vulnerabilities:** Databases are essential for web applications to efficiently manage large amounts of data accessible from multiple locations. While production environments typically utilize dedicated database, servers managed by software like MySQL or MariaDB, smaller applications might resort to storing data in flat-file databases. These databases are stored as single files on the computer, eliminating the complexity of setting up a dedicated server. However, this simplicity can introduce security vulnerabilities.

Consider a scenario where a flat-file database, potentially containing sensitive user information, is mistakenly stored within the web application's root directory. This reliability accessible location makes it possible for an attacker to download the database file and access its contents using a dedicated client like sqlite3 on their own machine. This exposes sensitive data, posing a significant security risk.

• **Simulation:**

In Fig. 3, A hypothetical scenario involves an attacker gaining access to a ‘/assets’ page and identifying a downloadable file named "webapp.db". This file, potentially containing sensitive database information, could be downloaded and analyzed using a SQLite database management tool like sqlite3. By examining the database schema using the .tables command and issuing appropriate SQL queries, the attacker could potentially retrieve all data stored within dataset.

Fig. 3. Exploiting Cryptographic Failure

2.3 A03:2021-Injection

• **Injection Vulnerabilities:** A Threat to Modern Applications: Injection vulnerabilities are a prevalent security threat in modern applications. These vulnerabilities arise when user supplied data is misinterpreted as code or commands by the application. The specific nature of the injection attack depends on the underlying technologies used by the application and how they handle user input.

• **Common Injection Vulnerabilities:**
SQL Injection (SQLi): This vulnerability occurs when untrusted user input is directly incorporated into SQL queries [4]. Malicious actors can exploit this weakness by crafting specially crafted input that injects malicious SQL code [5]. This injected code can then manipulate the intended query, potentially allowing attackers to:
Access sensitive data: Attackers can retrieve confidential information stored in the database, such as personal details and credentials.
Modify data: Malicious actors can alter data with-in the database, potentially causing disruption or corrupting critical information.
Delete data: Attackers can erase data stored in the database, leading to data loss and potential system instability.

• **Command Injection:** This vulnerability occurs when user-controlled input is passed directly to operating system commands. Attackers can exploit this weakness by injecting malicious commands that will be executed on the server. This can potentially grant them unauthorized access to the server's resources or even allow them to compromise the entire system. Attackers can leverage command injection to perform various malicious activities, including:

System Enumeration: They can identify files, directories, and running processes on the server.
Data Exfiltration: Attackers can steal sensitive data stored on the server.
Privilege Escalation: They can attempt to gain higher privileges on the server, potentially leading to complete system control.

• **Preventing Injection Attacks:** The primary defence against injection attacks is to ensure that user-supplied input is never treated as executable code or commands. Several techniques can achieve this:

Input Validation: Implement robust validation mechanisms to sanitize user input and remove any potentially dangerous characters before processing.
Parameterized Queries: Utilize parameterized queries instead of string concatenation when constructing database queries. This ensures clear separation between data and code, preventing malicious SQL injection attempts.
Escaping User Input: When user input must be included within commands or queries, implement proper escaping mechanisms to neutralize potentially harmful characters.
Whitelisting: Limit acceptable user input to a predefined set of safe characters or values. Any input that falls outside this whitelist should be rejected [6].
By implementing these security measures, developers can significantly reduce the risk of injection vulnerabilities in their applications [7].

• **Simulation:**
Fig, 4 shows to simulate a directory listing attack, the input field was injected with the string '$(ls)'. This attempted to exploit a potential command injection vulnerability by injecting a command to list directory contents. Upon submission, the response revealed the presence of an unexpected file named 'drpepper.txt' within the website's root directory, suggesting a potential security weakness.

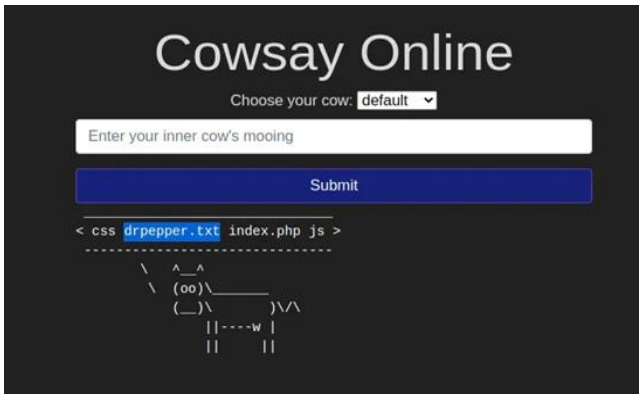

Fig. 4. Exploiting Injection

2.4 A04: Insecure Design:

Insecure design vulnerabilities are weaknesses embedded within an application's fundamental architecture. They differ from implementation or configuration flaws by being inherent to the core design concept. These vulnerabilities often stem from inadequate threat modeling during the application's planning phase, potentially impacting the entire application. Alternatively, insecure design flaws can be introduced by developers seeking shortcuts to streamline testing. For instance, a developer might disable two-factor authentication (2FA) during development for easier testing, neglecting to re-enable it before deployment [8].

• **Insecure Password Resets:** A Case Study: A classic example of insecure design is a vulnerability that once

existed on Instagram's password reset functionality. The platform relied on SMS delivery of a 6-digit code for password resets. An attacker could attempt to brute-force this code; however, Instagram implemented rate-limiting to prevent such attacks, blocking users after 250 attempts.

The critical design flaw here lies in the rate-limiting mechanism being restricted to individual IP addresses. An attacker with access to numerous IP addresses could circumvent this protection. With 250 attempts per IP and a million possible codes, an attacker would need approximately 4,000 IP addresses to cover all possibilities. While a large number, cloud services make acquiring such resources relatively inexpensive, rendering the attack viable.

This vulnerability highlights how insecure design flaws can arise from assumptions about user behavior. In this case, the design presumed users wouldn't have access to thousands of IP addresses. The issue stemmed from the core design, not the code implementation itself.

• **Addressing Insecure Design:** Due to their early introduction in the development process, resolving insecure design vulnerabilities often necessitates refactoring or rebuilding the affected application components. This makes them more challenging to rectify compared to traditional code-based vulnerabilities. The most effective approach to mitigating these risks involves thorough threat modeling during the initial development stages. You can explore resources like the Secure Software Development Lifecycle (SSDL) room for further guidance on implementing secure development practices.

Simulation Example:

To illustrate the design flaw in the password reset mechanism, let's revisit Joseph's account. By navigating to the password reset page, we can attempt to provide a guess for the security question answer. In a scenario where the security questions lack sufficient complexity or rely on easily obtainable information (e.g., favorite color), an attacker might successfully guess the answer. In this hypothetical example, by correctly guessing "green" as the answer to Joseph's security question in Fig. 5, we could potentially gain unauthorized access to his account. It's crucial to emphasize that this scenario highlights a vulnerability and is not a recommendation to exploit such weaknesses in real-world situations.

Fig. 5. Exploiting Insecure Design

2.5 A05: Security Misconfiguration

Security misconfigurations differ from other OWASP Top 10 vulnerabilities because they arise from improper configuration, even with up-to-date software. These misconfigurations can create exploitable weaknesses within systems [9].

Common examples of security misconfigurations include:

• **Improper Permission Management:** Inadequate access controls on cloud storage (e.g., overly permissive S3 bucket permissions) can expose sensitive data.

• **Unnecessary Features:** Leaving unused services, pages, accounts, or privileges enabled creates unnecessary attack surfaces.

• **Weak Default Credentials:** Failure to change default usernames and passwords creates easy entry points for attackers.

• **Information Leakage:** Excessively verbose error messages can unintentionally reveal sensitive system details to attackers.

• **Missing Security Headers:** Omission of essential HTTP security headers (e.g., Content Security Policy) leaves applications vulnerable to various attacks.

These misconfigurations can have cascading effects, potentially leading to vulnerabilities like:

• **Exploiting Default Credentials:** Gaining unauthorized access to sensitive data using unchanged default logins.

• **XML External Entity (XXE) Attacks:** Leveraging misconfigured XML parsers to inject malicious code for unauthorized access.

• **Command Injection Vulnerabilities:** Executing arbitrary commands on the system through vulnerable admin pages.

For a deeper understanding, refer to the OWASP Top 10 entry for Security Misconfiguration.

Debugging Interfaces

A prevalent security misconfiguration involves exposing debugging features in production environments. These features, intended for development purposes, offer advanced functionalities to developers but can be misused by attackers if left accessible.

Case Study: Patreon Hack (2015) [10]

The incident highlights the potential dangers of exposed debugging interfaces. In 2015, Patreon reportedly suffered a security breach allegedly linked to an open debugging interface. A security researcher had previously notified Patreon about a vulnerable Werkzeug console accessible via a URL path (/console).

Werkzeug, a core component in many Python web applications, provides a web server interface for executing Python code. It includes a built-in debug console accessible through a specific URL or automatically during application exceptions. Both scenarios present a Python console allowing attackers to execute arbitrary commands on the system, potentially compromising sensitive data or functionality.

This case study demonstrates the critical importance of disabling debugging features before deploying applications to production environments.

• **Simulation:**

This vulnerable machine demonstrates a Security Mis-configuration, a critical vulnerability listed in the Table 1. To exploit this misconfiguration in Fig. 6, we attempt to gain unauthorized access to the application's source code by navigating to the following URL: http://machine-ip/console

Fig. 6. Checking For python working or not

Our investigation reveals that the console is accessible without proper authentication, suggesting a security misconfiguration. To capitalize on this vulnerability, we execute a simple Python code snippet: import os; print (os.popen("ls -l").read()). This code successfully executes and displays the directory listing of the server, potentially revealing sensitive information such as file-names and permissions, as you can see from Fig 7.

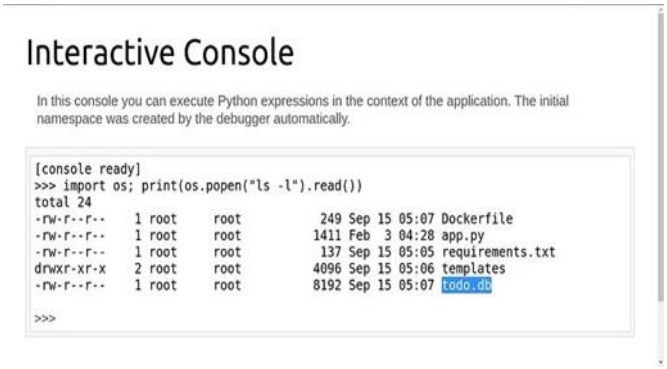

Fig. 7. Exploiting Security Misconfiguration

2.6 A06: Vulnerable and Outdated Components

The sixth vulnerability in Table 1 is "Vulnerable and Outdated Components" (A06:2021). This vulnerability arises when an application uses component such as libraries, frameworks, and other software modules that are outdated or have known security vulnerabilities. These components may come from open-source projects, third party vendors, or even from within the organization.

• **Outdated Versions:** Applications using components that are no longer supported or updated by their developers are prone to known vulnerabilities that attackers can easily exploit.

• **Unpatched Vulnerabilities**: When known security vulnerabilities in components are not patched, they become entry points for attackers, potentially leading to data breaches or system compromises.

• **Insecure Configuration:** Even updated components can be vulnerable if not configured securely, leading to potential exploitation.

• **Lack of Inventory Management:** Organizations may not have a comprehensive inventory of all components used in their applications, making it difficult to track and update them, thereby increasing security risks.

• **Transitive Dependencies:** Vulnerabilities may exist in dependencies that are indirectly included in the application, further complicating security management.

Prevention Strategies: To mitigate the risks associated with Vulnerable and Outdated Components, organizations can implement the following security measures:

• **Regular Updates:** Keep all software components, including libraries and frameworks, up to date with the latest security patches and versions to reduce the risk of exploitation.

• **Vulnerability Scanning:** Use automated tools to regularly scan for known vulnerabilities in your application’s dependencies and take immediate action to address any issues found.

• **Component Inventory**: Maintain a comprehensive inventory of all components, including direct and transitive

dependencies, to ensure that each component is tracked and updated as needed.

• **Automated Dependency Management**: Use automated tools for dependency management and security analysis to detect and mitigate vulnerabilities early in the development process.

• **Secure Configuration Practices**: Ensure that all components are configured securely according to best practices and that any default configurations are reviewed and hardened.

• **Risk Assessment of Dependencies**: Regularly assess the security risks associated with using specific components, especially those that are widely used or critical to the application's functionality.

By incorporating these preventive strategies, organizations can significantly reduce the risks associated with vulnerable and outdated components, thereby improving the overall security posture of their applications.

2.7 A07: Identification and Authentication Failures

Modern web applications rely heavily on robust authentication and session management mechanisms to ensure secure access for legitimate users. Authentication verifies user identities, typically through username and password combinations. Upon successful verification, the server issues a session cookie to the user's browser. This is necessary because HTTP(S) communication is stateless, requiring session cookies for the server to maintain user context and track user actions.

Vulnerabilities in Authentication Mechanisms:

Weaknesses in authentication mechanisms can be exploited by attackers, potentially leading to Broken Authentication (A07:2021) within the OWASP Top 10 (see Table 1). Some common vulnerabilities include:

• **Brute Force Attacks:** These attacks involve repeatedly attempting to guess usernames and passwords. Weak password policies and a lack of lockout mechanisms can make applications susceptible to brute force attacks [11].

• **Weak Credentials**: If web applications allow users to set weak passwords like "password1" or common dictionary words, attackers can easily guess them and gain unauthorized access. Applications should enforce strong password policies, including minimum length, character complexity, and regular password changes.

• **Weak Session Cookies:** Session cookies are how the server keeps track of users. If session cookies lack sufficient randomness or predictability in their values, attackers can potentially steal or forge them, enabling unauthorized access to user accounts.

Mitigation Strategies:

There can be various mitigation strategies for broken authentication mechanisms depending on the exact flaw. Here are some common approaches:

• **Enforce Strong Password Policies**: Minimum password length, character complexity (uppercase/lowercase letters, numbers, symbols), and regular password changes can significantly increase password strength.

• **Limit Login Attempts:** Implement lockout mechanisms that automatically lock user accounts after a certain number of failed login attempts. This thwarts brute force attacks by significantly increasing the number of attempts required for success.

• **Implement Multi-Factor Authentication (MFA**): MFA adds an additional layer of security by requiring users to provide a second authentication factor beyond a username and password. This could involve a code sent to a registered phone number, a fingerprint scan on a mobile device, or a hardware token.

By implementing these mitigation strategies, organizations can significantly strengthen their authentication mechanisms and reduce the risk of unauthorized access to web applications.

2.8 A08: Software and Data Integrity Failures

Software and Data Integrity Failures (A08:2021 in the OWASP Top 10) encompass vulnerabilities that arise when software and its underlying infrastructure lack proper mechanisms to safeguard against unauthorized modifications. This can manifest in various ways, such as:

• **Untrusted Dependencies:** Applications relying on plugins, libraries, or modules from untrusted sources, repositories, or content delivery networks (CDNs) create a vulnerability.

• **Insecure CI/CD Pipelines**: Weaknesses in the Continuous Integration and Continuous Delivery (CI/CD) pipeline can introduce vulnerabilities. These could involve unauthorized access, the injection of malicious code during the build process, or system compromise.

• **Unverified Auto-Updates:** Many applications now feature automatic update functionality. If these updates lack sufficient integrity verification before being applied, attackers can potentially upload their own malicious updates to compromise systems.

• **Insecure Deserialization:** When objects or data are encoded or serialized into a vulnerable structure, attackers can potentially exploit this weakness to modify the data and potentially gain unauthorized access.

Prevention Strategies: To mitigate Software and Data Integrity Failures, organizations can implement the following security measures:

• **Digital Signatures:** Utilize digital signatures or similar mechanisms to verify the authenticity and integrity of software or data. This ensures that the source is legitimate and the data has not been tampered with during transmission.
• **Trusted Repositories:** Ensure that libraries and dependencies are obtained from trusted repositories managed by reputable organizations. If the risk profile is high, consider hosting an internal, vetted repository for critical components.
• **Software Supply Chain Security Tools**: Leverage software supply chain security tools like OWASP Dependency Check or OWASP CycloneDX. These tools help identify and manage vulnerabilities within software components used by your application.
• **Code and Configuration Review:** Implement a thorough review process for code and configuration changes. This helps minimize the risk of introducing malicious code or insecure configurations into the software development pipeline.
• **CI/CD Pipeline Security:** Fortify the CI/CD pipeline by ensuring proper segregation of duties, secure configuration, and robust access controls. These measures safe-guard the integrity of code throughout the build and deployment processes.
• **Data Serialization Protection**: Avoid sending unsigned or unencrypted serialized data to untrusted clients. Implement integrity checks or digital signatures to detect any tampering or replay of sensitive serialized data.

By incorporating these preventive strategies, organizations can significantly enhance software and data integrity, reducing the attack surface for malicious actors.

2.9 A09: Security Logging and Monitoring Failures

The Importance of Web Application Logging and Monitoring: Web application security relies heavily on proper logging and monitoring practices. When a user interacts with a web application, every action performed should be meticulously logged. This data becomes invaluable in the event of a security incident, as it allows for:

Essential Log Information: To facilitate effective incident response and threat detection, application logs should capture critical information, including:

• **HTTP Status Codes**: These codes indicate the success or failure of a user request (e.g., 200 for successful requests, 404 for page not found errors).

• **Timestamps:** Time stamps provide a chronological record of user activity, aiding in incident timeline creation and attack sequence analysis.

• **Usernames**: Identifying the user associated with each action simplifies log analysis and helps assess potential compromised accounts.
• **API Endpoints/Page Locations:** Logging the specific web application resources accessed allows for a clear understanding of the attacker's target and potential areas of compromise.
• **IP Addresses:** Capturing the source IP address of each user request can help identify suspicious activity originating from unusual locations or known malicious actors.

Log Security and Retention: While logs contain valuable data, it's equally crucial to ensure their security. Sensitive information within logs should be encrypted at rest and in transit. Additionally, it's recommended to maintain multiple copies of logs in diverse locations for redundancy and disaster recovery purposes.

Beyond Logging: Implementing Security Monitoring: Although logging is critical for incident response and forensic analysis, it's most effective when coupled with real-time security monitoring practices. Security monitoring systems actively analyze log data and user activity to identify suspicious patterns that might suggest ongoing attacks. This proactive approach allows security teams to detect and potentially stop attackers before significant damage occurs.

Examples of Suspicious Activity:

Security monitoring systems can be configured to identify various indicators of potential threats, including:

• **Brute-Force Attacks:** Multiple failed login attempts within a short timeframe suggest brute-force attacks targeting user accounts.
• **Anomalous IP Addresses/Locations:** Access attempts originating from unusual locations or known malicious IP addresses warrant investigation.
• **Automated Tools:** Certain automated tools used by attackers can be identified based on patterns within user-agent headers or request speeds.
• **Exploit Attempts:** Security systems can be configured to detect known malicious payloads or exploit signatures within user requests.

Prioritizing Suspicious Activity:
Not all suspicious activity is equally concerning. Security monitoring systems should categorize alerts based on their potential impact. High-risk activities, such as attempts to access critical resources, should trigger immediate alerts and require swift response. Lower-risk incidents may necessitate further investigation but might not necessitate immediate action.

By implementing comprehensive logging and monitoring strategies, organizations can significantly enhance their web application security posture. Early detection and rapid response are paramount for mitigating the impact of security incidents and safeguarding sensitive data.

2.10 A10: Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) is a web application vulnerability that allows an attacker to manipulate a web application into making unauthorized requests to an external server under the attacker's control. This manipulation typically involves exploiting functionalities within the application that interact with external services [12].

Understanding the Vulnerability:

Imagine a web application that uses an external API to send SMS notifications to users. This application likely sends requests to the SMS provider's server with the message content and an authentication token (e.g., API key) to identify the sender. If the application allows user input to specify the server address of the SMS provider, a vulnerability can arise.

This would trick the vulnerable application into sending a request to the attacker's-controlled server at:

https://attacker.thm/api/send?msg=Test%20Message

As part of the forwarded message, the attacker might be able to steal the application's API key embedded in the request. This stolen key could then be used to send SMS messages at the application owner's expense.

Potential Impacts:

SSRF vulnerabilities can have various consequences depending on the application's functionalities and the attacker's goals. Here are some potential impacts:

- **Internal Network Enumeration:** Attackers can exploit SSRF to identify internal network addresses and ports, potentially aiding further attacks (see Fig 8).

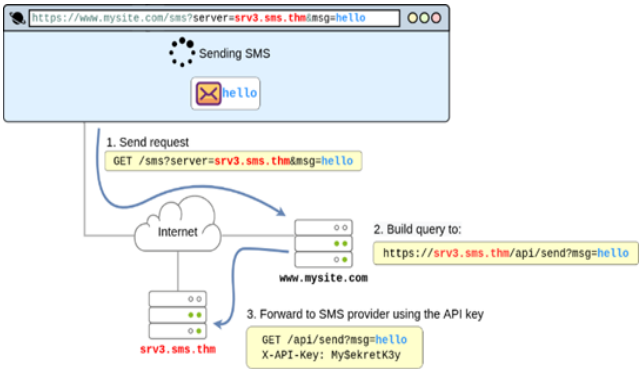

Fig. 8. Exploring Backend API Handling

- **Abuse of Trust Relationships:** SSRF can be used to exploit trust relationships between the application server and other internal services, potentially leading to unauthorized access to restricted resources.
- **Remote Code Execution (RCE):** In some cases, SSRF can be chained with other vulnerabilities to achieve remote code execution on the victim server, allowing complete control.

By understanding SSRF vulnerabilities and implementing proper security measures, organizations can significantly reduce the risk of unauthorized actions and data breaches.

3. Analysis of the OWASP Top 10 vulnerabilities

1. Exploitability: Exploitability in the context of vulnerability analysis, refers to the ease with which an attacker can leverage a specific vulnerability to gain unauthorized access to a system or data. It essentially reflects the technical difficulty and resources required for an attacker to successfully exploit the vulnerability.

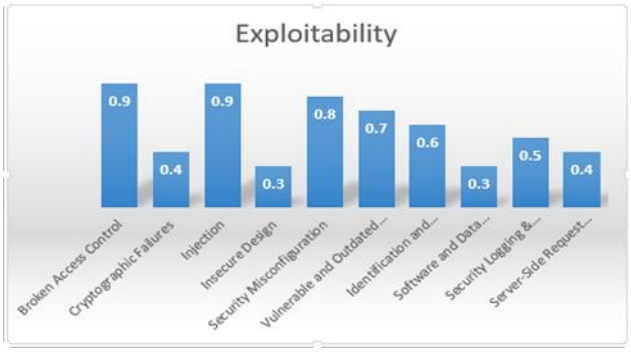

Fig. 9. Exploitability of vulnerabilities

From Fig. 9, a higher exploitability rating indicates a greater risk, as it suggests a wider range of attackers could potentially take advantage of the vulnerability. This is often factored into vulnerability scoring systems and prioritization for patching.

2. Incident Rate: This factor delves into the frequency of reported incidents associated with each vulnerability. A high incident rate indicates a vulnerability that is actively exploited by attackers in the real world. This information is crucial for prioritizing remediation efforts.

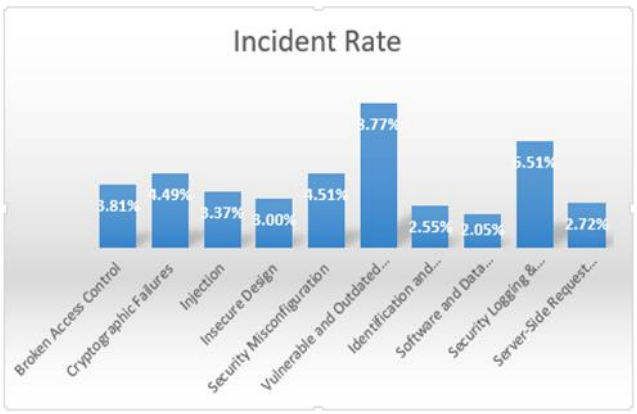

Fig. 10. Incident Rate of vulnerabilities

Fig. 10, it's crucial to remember that even a low incident rate doesn't guarantee a vulnerability is not dangerous. Attackers may be constantly evolving their tactics, and a seemingly obscure vulnerability could be weaponized in the future.

3. Criticality Ratio: This metric divides the exploitability score by the impact score, giving a higher value for vulnerabilities that are both easy to exploit and have a high impact. Criticality Ratio of a vulnerability refers to the level of risk it poses to a system or organization.

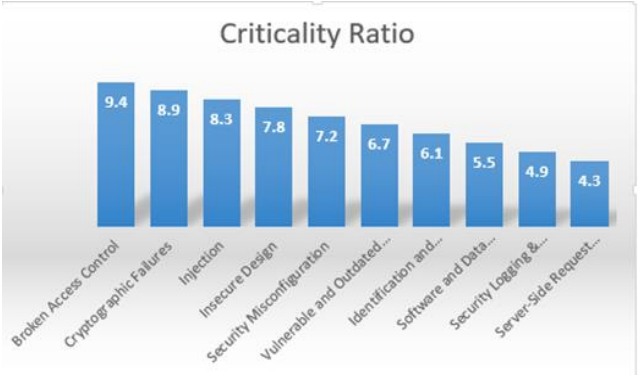

Fig. 11. Criticality Ratio of Vulnerabilities

Higher criticality indicates a more severe vulnerability (Fig. 11). This means it has the potential to cause significant damage and is also relatively easy for attackers to exploit. Conversely, a vulnerability with lower criticality might have a less severe potential impact or be more difficult to exploit, making it less urgent to address.

4. Conclusion

In today's digital landscape, web applications are the backbone of countless operations. However, these applications are vulnerable to exploitation by malicious actors, potentially leading to data breaches, disrupted services, and reputational damage. The OWASP Top 10 serves as a vital resource by identifying and categorizing the most critical web application security risks.

This report conducted an in-depth analysis of the top five vulnerabilities within the Table 1, providing a comprehensive examination of their technical details, potential consequences, and mitigation strategies. The remaining five vulnerabilities were covered with a basic introduction, highlighting their key characteristics and potential risks. By understanding the criticality ratio, incident rate, and exploitability for each vulnerability, organizations can prioritize their security efforts and address the most pressing threats.

By implementing robust security practices, leveraging the insights provided by the OWASP Top 10, and continuously monitoring and updating their defenses, organizations can significantly strengthen their web application security posture and safeguard their valuable assets.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] "OWASP Top Ten | OWASP Foundation." Accessed: Jul. 13, 2023. [Online]. Available: https://owasp.org/www-project-top-ten/

[2] M. M. Hassan, M. A. Ali, T. Bhuiyan, M. H. Sharif, and S. Biswas, "Quantitative Assessment on Broken Access Control Vulnerability in Web Applications", 2018.

[3] "OWASP Top 10 - 2021," TryHackMe. Accessed: Aug. 13, 2023. [Online]. Available: https://tryhackme.com/r/room/owasptop102021

[4] D. A. Kindy and A.-S. K. Pathan, "A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques," in 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE), Singapore, Singapore: IEEE, Jun. 2011, pp. 468–471. doi: 10.1109/ISCE.2011.5973873.

[5] S. Tyagi and K. Kumar, "Evaluation of Static Web Vulnerability Analysis Tools," in 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan Himachal Pradesh, India: IEEE, Dec. 2018, pp. 1–6. doi: 10.1109/PDGC.2018.8745996.

[6] I. Balasundaram and E. Ramaraj, "An Authentication Mechanism to prevent SQL Injection Attacks," International Journal of Computer Applications, vol. 19, 2011.

[7] "OWASP Top 10–2021 | Tryhackme Writeup/Walkthrough | By Md Amiruddin | by Md Amiruddin | InfoSec Write-ups." Accessed: Aug. 13, 2023. [Online]. Available: https://infosecwriteups.com/owasp-top-10-2021-tryhackme-writeup-walkthrough-by-md-amiruddin-913e477c0ea1

[8] B. Schneier, "Cryptographic design vulnerabilities," Computer, vol. 31, no. 9, pp. 29–33, Sep. 1998, doi: 10.1109/2.708447.

[9] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early Detection of Security Misconfiguration Vulnerabilities in Web Applications," in 2011 Sixth International Conference on Availability, Reliability and Security, Vienna, Austria: IEEE, Aug. 2011, pp. 169–174. doi: 10.1109/ARES.2011.31.

[10] Detectify, "How Patreon got hacked - Frans Rosén," Labs Detectify. Accessed: Aug. 13, 2023. [Online]. Available: https://labs.detectify.com/writeups/how-patreon-got-hacked-publicly-exposed-werkzeug-debugger/

[11] C. J. Mok and C. W. Chuah, “An Intelligence Brute Force Attack on RSA Cryptosystem,” vol. 1, no. 1, 2019.

[12] A. Younis, Y. K. Malaiya, and I. Ray, “Assessing vulnerability exploitability risk using software properties,” Software Qual J, vol. 24, no. 1, pp. 159–202, Mar. 2016, doi: 10.1007/s11219-015-9274-6.